Domain-Driven Design

Domain-Driven Design

- Domain-Driven Design is a concept introduced by a programmer **Eric Evans** in 2004 in his book **Domain-Driven Design: Tackling Complexity in Heart of Software**.
- Domain-Driven Design is an approach to software design that glues the system's implementation to a constantly evolving model, leaving aside irrelevant details like programming languages, infrastructure technologies, etc
- **What is Domain ?**
  -The word Domain used in context of software development refers to *business*.
- *When we are developing software our focus should not be primarily on technology, rather it should be primarily on business*.

Credits: https://dzone.com/refcardz/getting-started-domain-driven

- Domain-Driven Design is an approach to software development that centers the development on programming a domain model that has a rich understanding of the processes and rules of a domain.

- Domain-Driven Design is all about design and creating highly expressive models. DDD also aims to create models that are understandable by everyone involved in the software development, not just software developers.

- Since non-technical people also work with these models, it is convenient if the models can be represented in different ways. Typically, a model of a domain can be depicted as a UML sketch, as code, and in the language of the domain.

Credits: https://dzone.com/refcardz/getting-started-domain-driven

- **Using Language**

“A person that is looking at attending a training course searches for courses based on topic, cost and the course schedule. When a course is booked, a registration is issued which the person can cancel or accept at a later date.”

Credits: https://dzone.com/refcardz/getting-started-domain-driven

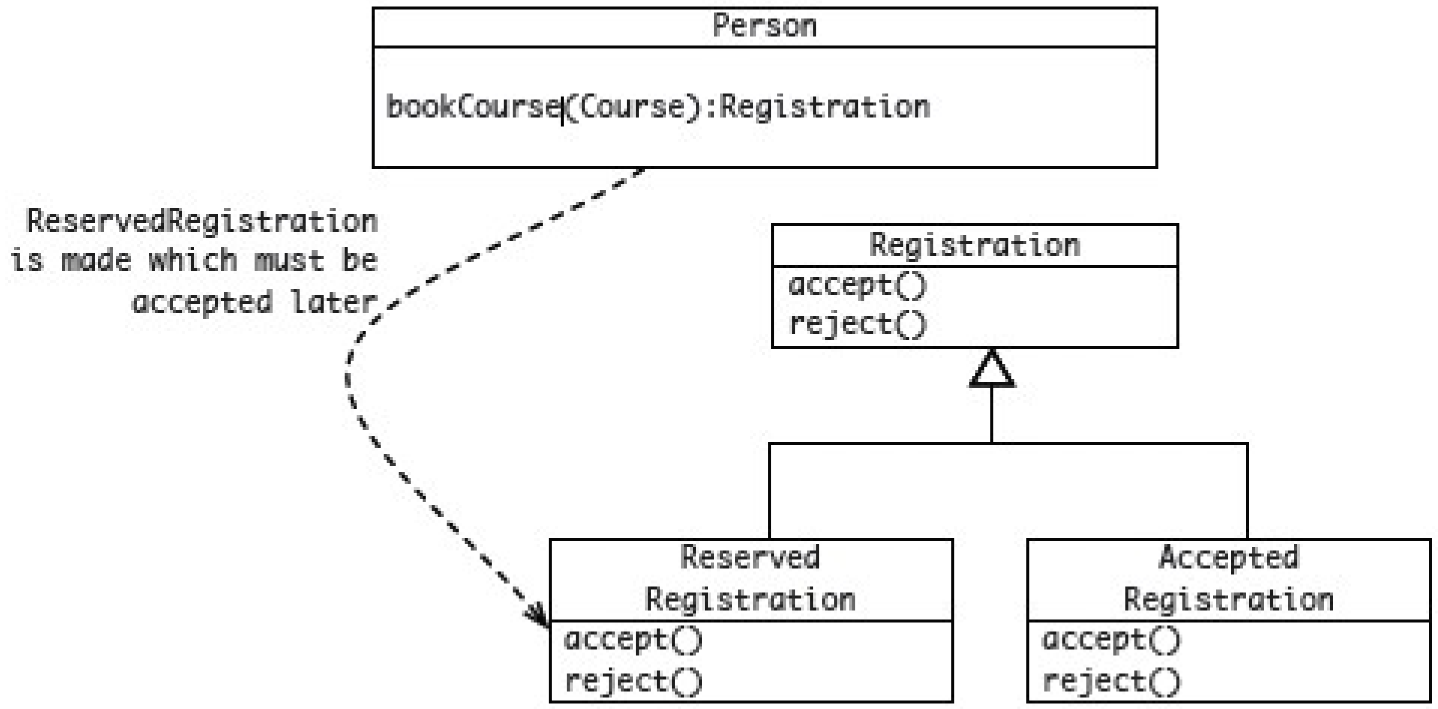
Using Code:

```java
class Person {
   public Registration bookCourse(Course c) { ' }
}
abstract class Registration {
    public abstract void accept();
    public abstract void cancel();
}

class ReservedRegistration extends Registration { ' }
class AcceptedRegistration extends Registration { ' }
interface CourseRepository {
    public List<Course> find(');
}
```

Credits: https://dzone.com/refcardz/getting-started-domain-driven

- **Using a UML Sketch**



Credits: https://dzone.com/refcardz/getting-started-domain-driven

Ubiquitous Language

- Ubiquitous Language is the term Eric Evans uses in Domain Driven Design for the practice of building up a common, rigorous language between developers and users.

- The consistent use of unambiguous language is essential in understanding and communicating insights discovered in the domain.

- In DDD, it is less about the nouns and verbs and more about the concepts.

Credits: https://dzone.com/refcardz/getting-started-domain-driven
https://martinfowler.com/bliki/DomainDrivenDesign.html

Ubiquitous Language

- Evans makes clear that using the ubiquitous language in conversations with domain experts is an important part of testing it, and hence the domain model.
- He also stresses that the language (and model) should evolve as the team's understanding of the domain grows.
- When you work with a ubiquitous language, the collaboration with domain experts is more creative and valuable for everyone.

Credits: https://dzone.com/refcardz/getting-started-domain-driven
https://martinfowler.com/bliki/DomainDrivenDesign.html
https://dannorth.net/whats-in-a-story/

Strategic Design

- Strategic design is about design in the large, and helps focus on the many parts that make up the large model, and how these parts relate to each other.

- In DDD, these smaller models reside in bounded contexts. The manner in which these bounded contexts relate to each other is known as context mapping.

Credits: https://dzone.com/refcardz/getting-started-domain-driven
https://martinfowler.com/bliki/DomainDrivenDesign.html
https://dannorth.net/whats-in-a-story/

Bounded Context

- **Bounded Context**: A description of a boundary (typically a subsystem, or the work of a specific team) within which a particular model is defined and applicable.

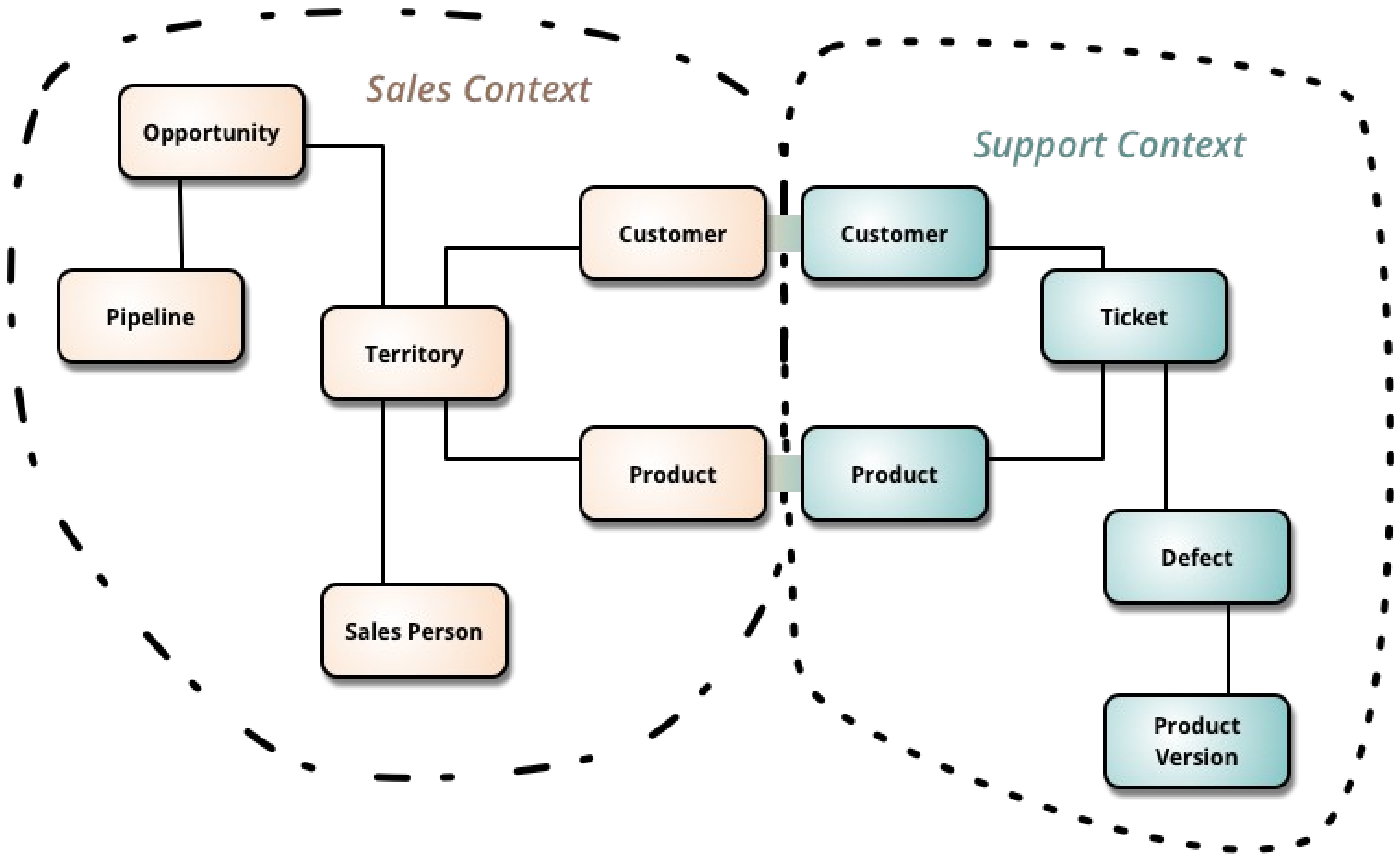
- Bounded Context is a central pattern in Domain-Driven Design.

- It is the focus of DDD's strategic design section which is all about dealing with large models and teams.

- DD deals with large models by dividing them into different Bounded Contexts and being explicit about their interrelationships

https://martinfowler.com/bliki/BoundedContext.html

Bounded Context



ttps://martinfowler.com/bliki/BoundedContext.html

Bounded Context

- In electricity utility - here the word "meter" meant subtly different things to different parts of the organization:
  - was it the connection between the grid and a location,
  - the grid and a customer,
  - the physical meter itself (which could be replaced if faulty).

- This confusion recur with polysemes like "Customer" and "Product".

ttps://martinfowler.com/bliki/BoundedContext.html

Bounded Contexts

- For each model, deliberately and explicitly define the context in which it exists. There are no rules to creating a context, but it is important that everyone understands the boundary conditions of the context.

- Contexts can be created from (but not limited to) the following:
  - how teams are organized
  - the structure and layout of the code base
  - usage within a specific part of the domain

- Aim for consistency and unity inside the context and don't be distracted by how the model is used outside the context.

- Other contexts will have different models with different concepts.

- It is not uncommon for another context to use a different dialect of the domain's ubiquitous language.

Context Maps

- Context mapping is a design process where the contact points and translations between bounded contexts are explicitly mapped out.
- Patterns:
- **Shared Kernel**
  - This is a bounded context that is a subset of the domain that different teams agree to share. It requires really good communication and collaboration between the teams.
- **Customer/Supplier Development Teams**
  - When one bounded context serves or feeds another bounded context, then the downstream context has a dependency on the upstream context. Knowing which context is upstream and downstream makes the role of supplier (upstream) and customer (downstream) explicit.
- **Conformist**
  - When the team working with the downstream context has no influence or opportunity to collaborate with the team working on the upstream context, then there is little option but to conform to the upstream context.

tps://dzone.com/refcardz/getting-started-domain-driven

Patterns: Context Mapping

- **Anti-corruption Layer**
  - When contexts exist in different systems and attempts to establish a relationship result in the 'bleeding' of one model into the other model, then the intention of both will be lost in the mangled combination of the models from the two contexts.
  - In this case, it is better to keep the two contexts well apart and introduce an isolating layer in-between that is responsible for translating in both directions.
  - This anti-corruption layer allows clients to work in terms of their own models.
- **Separate Ways**
  - Critically analyze the mappings between bounded contexts. If there are no indispensable functional relationships, then keep the context separate.

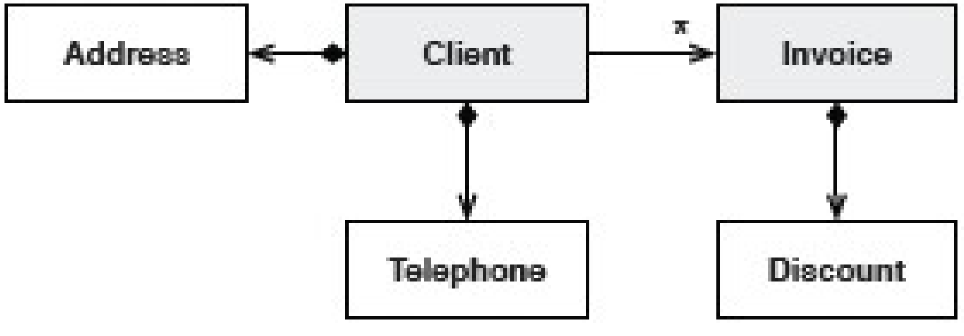
tps://dzone.com/refcardz/getting-started-domain-driven

Modeling the Domain

- **Dealing with Structure**
  - Entities
  - Value Objects
  - Cardinality of Associations
  - Services
  - Aggregates

Entities

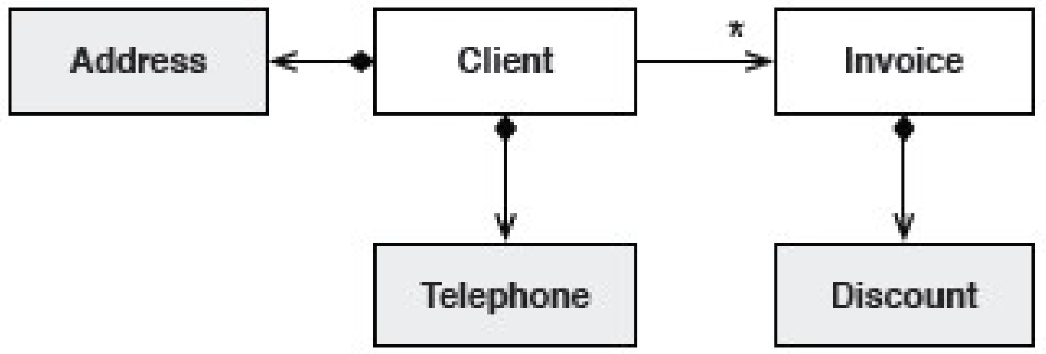
- Entities are classes where the instances are globally identifiable and keep the same identity for life. There can be change of state in other properties, but the identity never changes.



tps://dzone.com/refcardz/getting-started-domain-driven

Value Objects

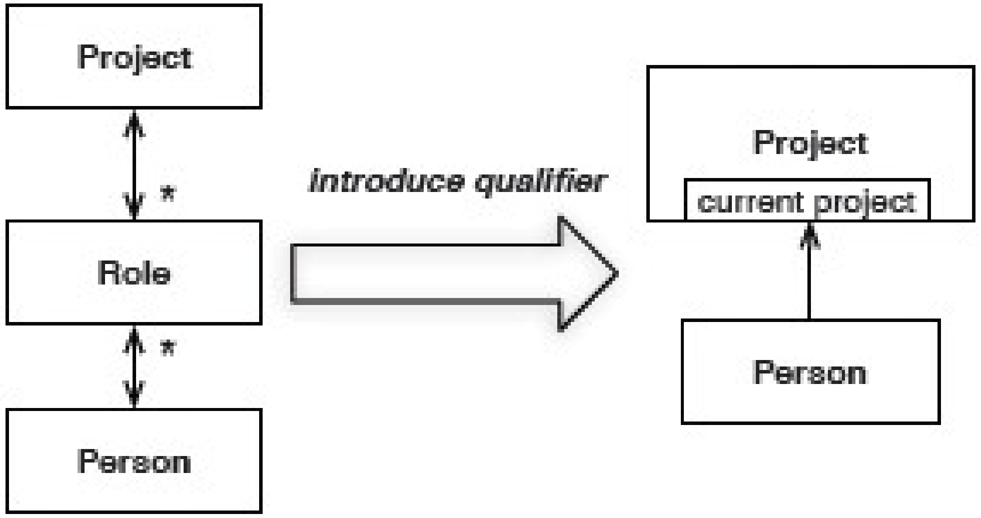
- Value objects are lightweight, immutable objects that have no identity.
- While their values are more important, they are not simple data transfer objects.
- Value objects are a good place to put complex calculations, offloading heavy computational logic from entities.
- They are much easier and safer to compose and by offloading heavy computational logic from the entities, they help entities focus on their role of life-cycle trackers.



tps://dzone.com/refcardz/getting-started-domain-driven

Cardinality of Associations

- The greater the cardinality of associations between classes, the more complex the structure. Aim for lower cardinality by adding qualifiers.
- Bi-directional associations also add complexity. Critically ask questions of the model to determine if it is absolutely essential between two objec



tps://dzone.com/refcardz/getting-started-domain-driven

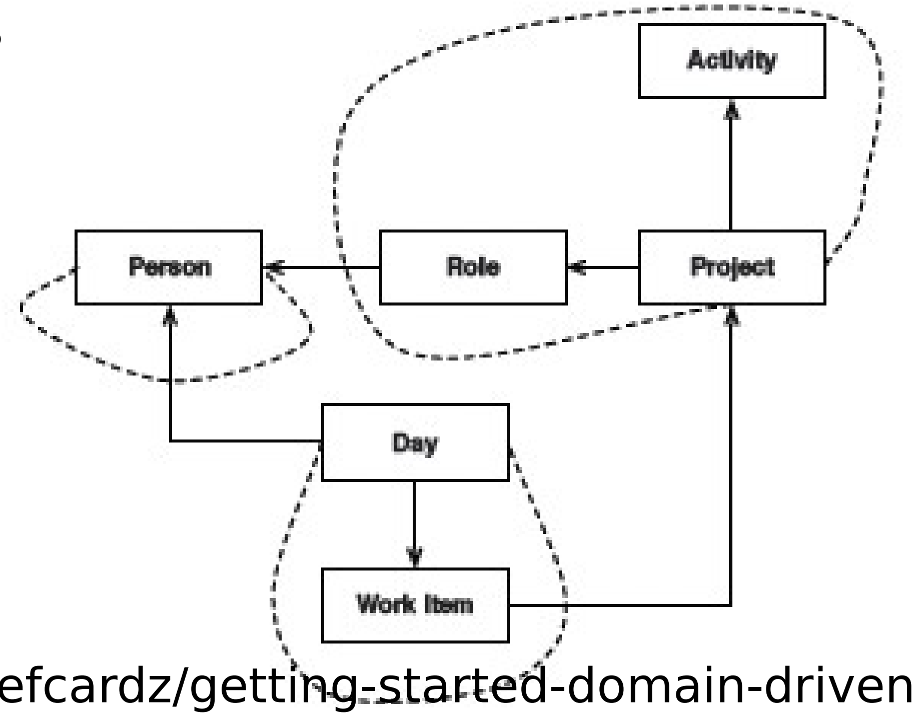
Services

- Sometimes it is impossible to allocate behaviour to any single class, be it an entity or value object.
- These are cases of pure functionality that act on multiple classes without one single class taking responsibility for the behaviour.
- In such cases, a stateless class, called a service class, is introduced to encapsulate this behaviour.

tps://dzone.com/refcardz/getting-started-domain-driven

Aggregates

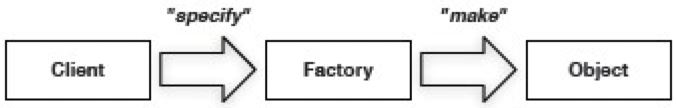
- Aggregates are consistency boundaries such that the classes inside the boundary are 'disconnected' from the rest of the object graph.
- Each aggregate has one entity which acts as the 'root' of the aggregate.
- When creating aggregates, ensure that the aggregate is still treated as a unit that is meaningful in the domain.
- Also, tes oundary by applying



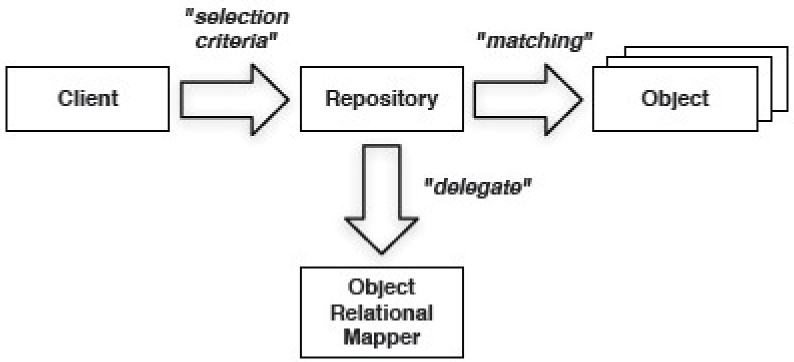
tps://dzone.com/refcardz/getting-started-domain-driven

Dealing with Life Cycles

- **Factories**: Factories manage the beginning of the life cycle of some aggregates.

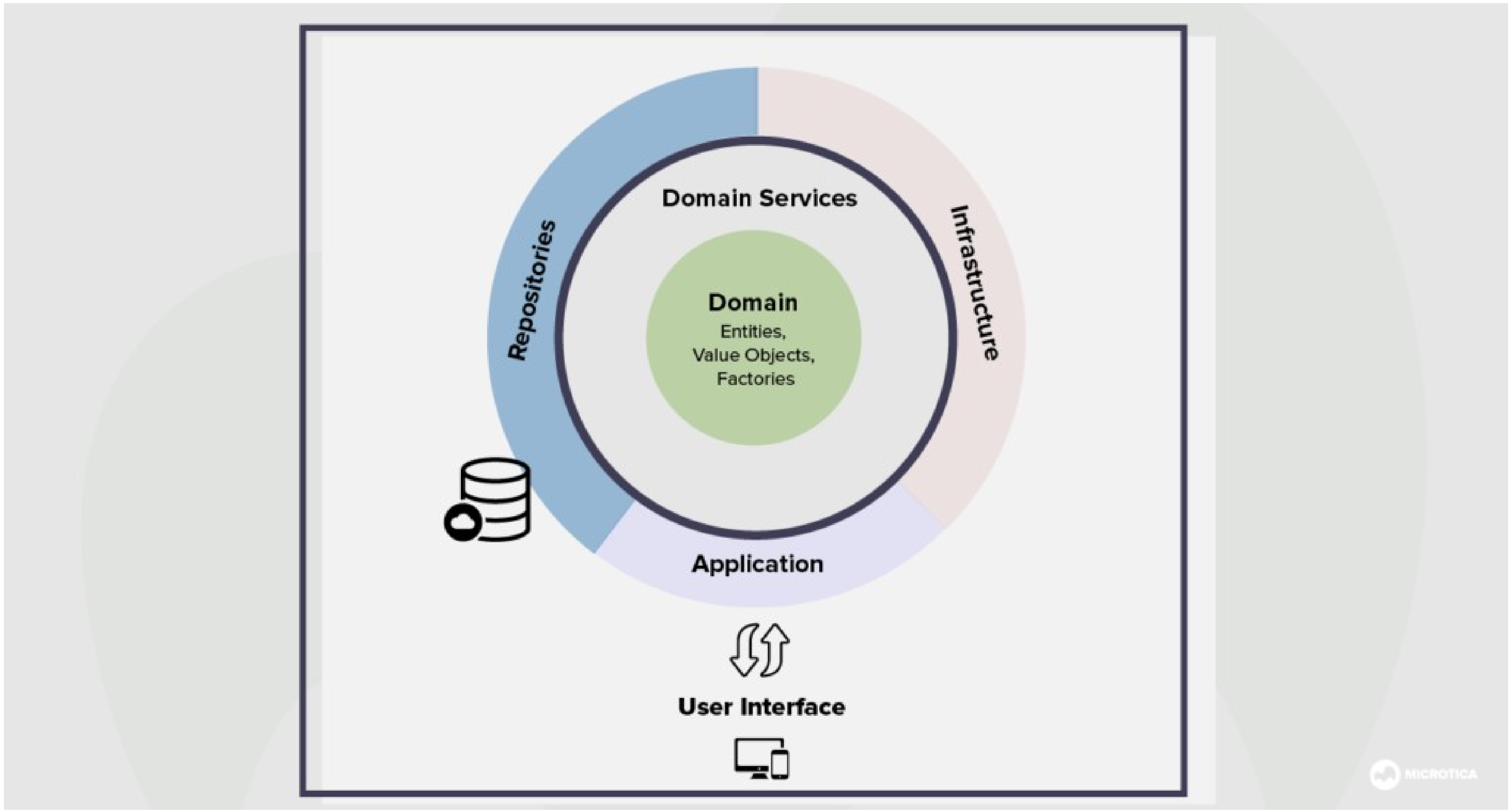


- **Repositories:** While factories manage the start of the life cycle, repositories manage the middle and end of the life cycle.
- Repositories might delegate persistence responsibilities to object-relational mappers for retrieval of objects.



tps://dzone.com/refcardz/getting-started-domain-driven

Example



//medium.com/microtica/the-concept-of-domain-driven-design-explained-3184c0

Advantages and Downsides of DDD

Advantages of Domain-Driven Design

- Simpler communication.
- More flexibility
- The domain is more important than UI/UX.

Downsides of Domain-Driven Design

- Deep domain knowledge is needed
- Contains repetitive practices.
- It might not work best for highly-technical projects

References

- Domain-Driven Design Tackling Complexity in the Heart of Software.
- https://dzone.com/refcardz/getting-started-domain-driven
- https://martinfowler.com/bliki/DomainDrivenDesign.html
- https://dannorth.net/whats-in-a-story/
- https://dzone.com/refcardz/getting-started-domain-driven
- https://medium.com/ssense-tech/domain-driven-design-everything-you-always-wanted-to-know-about-it-but-were-afraid-to-ask-a85e7b74497a
- https://www.ibm.com/garage/method/practices/code/domain-driven-design/
- https://medium.com/microtica/the-concept-of-domain-driven-design-explained-3184c0fd7c3f